Conservative use of `noexcept` in the Library

Document: N3279=11-0049
Date: 2011-03-25
Authors: Alisdair Meredith (ameredith1@bloomberg.net)
John Lakos (jlakos@bloomberg.net)

Motivation

The paper N3248 raised a number of concerns with widespread adoption of `noexcept` exception specifications in the standard library specification, preferring their use be left as a library vendor quality-of-implementation feature until we have more experience.

Further discussion at the Madrid meeting, 2011, showed that while the committee shared some of these concerns, it also wanted to promote the use of such exception specifications where they provided a benefit, and did no harm.

After some discussion, the following set of guidelines for appropriate use of `noexcept` in a library specification were adopted. The rest of this paper applies these guidelines to the working paper N3242.

Narrow and Wide Contracts

To ease the following discussion, we define two terms of art, 'narrow contract' and 'wide contract'. These terms are widely used with differing informal definitions, so we will define precisely what they mean (in this document) to avoid confusion.

Wide Contracts

A wide contract for a function or operation does not specify any undefined behavior. Such a contract has no preconditions: A function with a wide contract places no additional runtime constraints on its arguments, on any object state, nor on any external global state. Examples of functions having wide contracts would be `vector<T>::begin()` and `vector<T>::at(size_type)`. Examples of functions not having a wide contract would be `vector<T>::front()` and `vector<T>::operator[](size_type)`.

Narrow Contracts

A narrow contract is a contract which is not wide. Narrow contracts for a functions or operations result in undefined behavior when called in a manner that violates the documented contract. Such a contract specifies at least one precondition involving its arguments, object state, or some external global state, such as the initialization of a static object. Good examples of standard functions with narrow contracts are `vector<T>::front()` and `vector<T>::operator[](size_type)`.

Adopted Guidelines

- No library destructor should throw. They shall use the implicitly supplied (non-throwing) exception specification.
- Each library function having a **wide** contract, that the LWG agree cannot throw, should be marked as unconditionally `noexcept`.
- If a library swap function, move-constructor, or move-assignment operator is conditionally-wide (i.e. can be proven to not throw by applying the noexcept operator) then it should be marked as conditionally noexcept. No other function should use a conditional noexcept specification.
- Library functions designed for compatibility with "C" code (such as the atomics facility), may be marked as unconditionally `noexcept`.

Proposed Library Changes

This paper reviews only the clauses 17-28. Clauses 29 and 30 will be handled separately.

17.6.5.12 [res.on.exception.handling]

4 Destructor operations defined in the C++ standard library shall not throw exceptions. Every destructor in the standard library shall behave as if it had a non-throwing exception-specification. Any other functions defined in the C++ standard library that do not have an exception-specification may throw implementation-defined exceptions unless otherwise specified.190 An implementation may strengthen this implicit exception-specification by adding an explicit one.191

18.7.1 Class type_info [type.info]

```
namespace std {
  class type_info {
  public:
    virtual ~type_info() ~~noexcept~~;
    bool operator==(const type_info& rhs) const noexcept;
    bool operator!=(const type_info& rhs) const noexcept;
    bool before(const type_info& rhs) const noexcept;
    size_t hash_code() const noexcept;
    const char* name() const noexcept;

    type_info(const type_info& rhs) = delete;          // cannot be copied
    type_info& operator=(const type_info& rhs) = delete; // cannot be copied
  }
};
```

const char* name() const noexcept;

Returns: An implementation-defined ntbs.
Remarks: The message may be a null-terminated multibyte string (17.5.2.1.4.2), suitable for conversion and display as a wstring (21.3, 22.4.1.4)

20.3.2 Class template pair [pairs.pair]

```
// defined in header <utility>
namespace std {
  template <class T1, class T2>
    struct pair {
      typedef T1 first_type;
      typedef T2 second_type;

      T1 first;
      T2 second; pair(const pair&) = default;
      pair(pair&&) = default;
      constexpr pair();
      pair(const T1& x, const T2& y);
      template<class U, class V> pair(U&& x, V&& y) ~~noexcept(see below)~~;
      template<class U, class V> pair(const pair<U, V>& p);
      template<class U, classV> pair(pair<U, V>&& p) ~~noexcept(see below)~~;
      template <class... Args1, class... Args2>
        pair(piecewise_construct_t,
        tuple<Args1...> first_args, tuple<Args2...> second_args) ~~noexcept(see below)~~;

      pair& operator=(const pair& p);
      template<class U, class V> pair& operator=(const pair<U, V>& p);
      pair& operator=(pair&& p) noexcept(see below);
      template<class U, class V> pair& operator=(pair<U, V>&& p) ~~noexcept(see below)~~;

      void swap(pair& p) noexcept(see below);
  };
}
```

template<class U, class V> pair(U&& x, V&& y) ~~noexcept~~;
~~Remarks: The expression inside noexcept is equivalent to:~~
~~is_nothrow_constructible<T1, U&&>::value &&~~
~~is_nothrow_constructible<T2, V&&>::value~~

template<class U, class V> pair(pair<U, V>&& p) ~~noexcept~~;
~~Remarks: The expression inside noexcept is equivalent to:~~

~~is_nothrow_constructible<T1, U&&>::value &&~~
~~is_nothrow_constructible<T2, V&&>::value~~

template<class... Args1, class... Args2>
  pair(piecewise_construct_t,
    tuple<Args1...> first_args, tuple<Args2...> second_args) ~~noexcept~~;
~~17 Remarks: The expression inside noexcept is equivalent to:~~
~~is_nothrow_constructible<T1, Args1&&...>::value &&~~
~~is_nothrow_constructible<T2, Args2&&...>::value~~

template<class U, class V> pair& operator=(pair<U, V>&& p) ~~noexcept~~;
~~Remarks: The expression inside noexcept is equivalent to:~~
~~is_nothrow_assignable<T1&, U&&>::value &&~~
~~is_nothrow_assignable<T2&, V&&>::value~~

20.3.3 [pairs.spec]

template <class T1, class T2>
  pair<V1, V2> make_pair(T1&& x, T2&& y) ~~noexcept~~;
~~The expression inside noexcept is equivalent to:~~
~~is_nothrow_constructible<V1, T1&&>::value &&~~
~~is_nothrow_constructible<V2, T2&&>::value~~

20.4.2 Class template tuple

namespace std {
  template <class... Types>
  class tuple {
  public:

    // 20.4.2.1, tuple construction
    constexpr tuple();
    explicit tuple(const Types&...);
    template <class... UTypes>
      explicit tuple(UTypes&&...) ~~noexcept~~;

    tuple(const tuple&) = default;
    tuple(tuple&&) = default;
    template <class... UTypes> tuple(const tuple<UTypes...>&);
    template <class... UTypes> tuple(tuple<UTypes...>&&) ~~noexcept~~;

    template <class U1, class U2>
      tuple(const pair<U1, U2>&); // iff sizeof...(Types) == 2
    template <class U1, class U2>
      tuple(pair<U1, U2>&&) ~~noexcept~~; // iff sizeof...(Types) == 2

    // 20.4.2.2, tuple assignment
    tuple& operator=(const tuple&);
    tuple& operator=(tuple&&) noexcept(see below);

    template <class... UTypes>
      tuple& operator=(const tuple<UTypes...>&);
    template <class... UTypes>
      tuple& operator=(tuple<UTypes...>&&) ~~noexcept~~;
  };
}

20.4.2.1[tuple.cnstr]

template <class... UTypes>
  explicit tuple(UTypes&&... u) ~~noexcept~~;
~~Remark: The expression inside noexcept is equivalent to the logical and of the following expressions:~~
~~is_nothrow_constructible<Ti, Ui&&>::value~~
~~where Ti is the ith type in Types and Ui is the ith type in UTypes.~~

template <class... UTypes>
  tuple(tuple<UTypes...>&& u) ~~noexcept~~;
~~Remark: The expression inside noexcept is equivalent to the logical and of the following expressions:~~
~~is_nothrow_constructible<Ti, Ui&&>::value~~
~~where Ti is the ith type in Types and Ui is the ith type in UTypes.~~

template <class U1, class U2>
  tuple(pair<U1, U2>&& u) ~~noexcept~~;
25 ~~Remark: The expression inside noexcept is equivalent to:~~

~~is_nothrow_constructible<T1, U1&&>::value &&~~
~~is_nothrow_constructible<T2, U2&&>::value~~
~~where T1 and T2 are the first and second types, respectively, in Types.~~

20.4.2.2 [tuple.assign]
tuple& operator=(tuple&& u) noexcept(see below);
Remark: The expression inside noexcept is equivalent to the logical and of the following expressions:
is_nothrow_move_assignable<T_i>::value
where T_i is the i_{th} type in Types.

template <class... UTypes>
tuple& operator=(tuple<UTypes...>&& u) ~~noexcept~~;
~~Remark: The expression inside noexcept is equivalent to the logical and of the following expressions:~~
~~is_nothrow_assignable<Ti&, Ui&&>::value~~
~~where Ti is the ith type in Types and Ui is the ith type in UTypes.~~

template <class U1, class U2>
tuple& operator=(pair<U1, U2>&& u) ~~noexcept~~;
~~The expression inside noexcept is equivalent to:~~
~~is_nothrow_assignable<T1&, U1&&>::value &&~~
~~is_nothrow_assignable<T2&, U2&&>::value~~
~~where T1 and T2 are the first and second types, respectively, in Types.~~

20.5 [template.bitset]

```
namespace std {
  template<size_t N>
  class bitset {
  public:

    // ...
    // element access:
    constexpr bool operator[](size_t pos) const ~~noexcept~~;
    reference operator[](size_t pos) ~~noexcept~~;
  };
}
```

20.5.2 [bitset.members]

constexpr bool operator[](size_t pos) ~~noexcept~~;
49 *Requires*: pos shall be valid.
Throws: nothing.
50 *Returns*: true if the bit at position pos in *this has the value one, otherwise false.

bitset<N>::reference operator[](size_t pos) ~~noexcept~~;
51 *Requires*: pos shall be valid.
Throws: nothing.

20.6.2 [memory.syn]
Header <memory> synopsis

// 20.9.10.5, shared_ptr atomic access:

template<class T>
bool atomic_is_lock_free(const shared_ptr<T>* p) ~~noexcept~~;
template<class T> shared_ptr<T> atomic_load(const shared_ptr<T>* p) ~~noexcept~~;
template<class T> shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo) ~~noexcept~~;
template<class T> void atomic_store(shared_ptr<T>* p, shared_ptr<T> r) ~~noexcept~~;
template<class T> void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo) ~~noexcept~~;
template<class T> shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r) ~~noexcept~~;
template<class T> shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r,
memory_order mo) ~~noexcept~~;
template<class T> bool atomic_compare_exchange_weak(
shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w) ~~noexcept~~;
template<class T> bool atomic_compare_exchange_strong(
shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w) ~~noexcept~~;
template<class T> bool atomic_compare_exchange_weak_explicit(
shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w, memory_order success, memory_order failure) ~~noexcept~~;
template<class T> bool atomic_compare_exchange_strong_explicit(
shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w, memory_order success, memory_order failure) ~~noexcept~~;

// 20.9.11, Pointer safety
enum class pointer_safety { relaxed, preferred, strict }; void declare_reachable(void *p); template <class T> T
*undeclare_reachable(T *p) ~~noexcept~~;
void declare_no_pointers(char *p, size_t n) ~~noexcept~~;
void undeclare_no_pointers(char *p, size_t n) ~~noexcept~~;
pointer_safety get_pointer_safety() noexcept;

20.6.3 Pointer traits [pointer.traits]

The class template pointer_traits supplies a uniform interface to certain attributes of pointer-like types.

```
namespace std {
  template <class Ptr> struct pointer_traits {
    typedef Ptr pointer;
    typedef see below element_type;
    typedef see below difference_type;

    template <class U> using rebind = see below;

    static pointer pointer_to(see below r);
  };

  template <class T> struct pointer_traits<T*> {
    typedef T*  pointer;
    typedef T   element_type;
    typedef ptrdiff_t difference_type;

    template <class U> using rebind = U*;

    static pointer pointer_to(see below r) noexcept;
  };
}
```

20.6.3.2 Pointer traits member functions [pointer.traits.functions]

static pointer pointer_traits::pointer_to(see below r);
static pointer pointer_traits<T*>::pointer_to(see below r) noexcept;

Remark: if element_type is (possibly cv-qualified) void, the type of r is unspecified; otherwise, it is T&.

Returns: The first template function returns a dereferenceable pointer to r obtained by calling Ptr::pointer_to(r); an instantiation of this function is ill-formed if Ptr does not have a matching pointer_to static member function. The second template function returns std::addressof(r).

20.6.4 [util.dynamic.safety]

A complete object is declared reachable while the number of calls to declare_reachable with an argument referencing the object exceeds the number of calls to undeclare_reachable with an argument referencing the object.

void declare_reachable(void *p);
Requires: p shall be a safely-derived pointer (3.7.4.3) or a null pointer value.
Effects: If p is not null, the complete object referenced by p is subsequently declared reachable (3.7.4.3).
Throws: May throw std::bad_alloc if the system cannot allocate additional memory that may be required to track objects declared reachable.

template <class T>
  T *undeclare_reachable(T *p) ~~noexcept~~;
Requires: If p is not null, the complete object referenced by p shall have been previously declared reachable, and shall be live (3.8) from the time of the call until the last undeclare_reachable(p) call on the object.
Returns: a safely derived copy of p which shall compare equal to p.
Throws: nothing.
[Note: It is expected that calls to declare_reachable(p) will consume a small amount of memory in addition to that occupied by the referenced object until the matching call to undeclare_reachable(p) is encountered. Long running programs should arrange that calls are matched. — end note]

void declare_no_pointers(char *p, size_t n) ~~noexcept~~;
Requires: No bytes in the specified range have been previously registered with declare_no_pointers(). If the specified range is in an allocated object, then it must be entirely within a single allocated object. The object must be live until the corresponding undeclare_no_pointers() call. [Note: In a garbage-collecting implementation, the fact that a region in an object is registered with declare_no_- pointers() should not prevent the object from being collected. — end note]

Effects: The n bytes starting at p no longer contain traceable pointer locations, independent of their type. Hence pointers located there may not be dereferenced if the object they point to was created by global operator new and not previously declared reachable. [Note: This may be used to inform a garbage collector or leak detector that this region of memory need not be traced. — end note]

Throws: nothing.
[Note: Under some conditions implementations may need to allocate memory. However, the request can be ignored if memory allocation fails. — end note]

void undeclare_no_pointers(char *p, size_t n) ~~noexcept~~;
Requires: The same range must previously have been passed to declare_no_pointers(). Effects: Unregisters a range registered with declare_no_pointers() for destruction. It must be called before the lifetime of the object ends.
Throws: nothing.

pointer_safety get_pointer_safety() noexcept;
Returns: pointer_safety::strict if the implementation has strict pointer safety (3.7.4.3). It is im- plementation defined whether get_pointer_safety returns pointer_safety::relaxed or pointer_- safety::preferred if the implementation has relaxed pointer safety.

20.6.5 [ptr.align]

void *align(std::size_t alignment, std::size_t size, void *&ptr, std::size_t& space) ~~noexcept~~;

20.6.8 [allocator.traits]

```
namespace std {
  template <class Alloc> struct allocator_traits {
    // ...
    static void deallocate(Alloc& a, pointer p, size_type n) ~~noexcept~~;
  };
}
```

20.6.8.2 [allocator.traits.members]

static void deallocate(Alloc& a, pointer p, size_type n) ~~noexcept~~;

3 *Effects*: calls a.deallocate(p, n).
Throws: nothing.

20.6.9 [default.allocator]

```
namespace std {
  // ...

  template <class Alloc>class allocator {
    // ...
    void deallocate(pointer p, size_type n) ~~noexcept~~;
  };
}
```

20.6.10 [storage.iterator]

1 raw_storage_iterator is provided to enable algorithms to store their results into uninitialized memory. The formal template parameter OutputIterator is required to have its operator* return an object for which operator& is defined and returns a pointer to T, and is also required to satisfy the requirements of an output iterator (24.2.4).

```
namespace std {
  template <class OutputIterator, class T>
  class raw_storage_iterator
    : public iterator<output_iterator_tag,void,void,void,void> {
  public:
    explicit raw_storage_iterator(OutputIterator x) ~~noexcept~~;

    raw_storage_iterator<OutputIterator,T>& operator*() ~~noexcept~~;
    raw_storage_iterator<OutputIterator,T>& operator=(const T& element);
    raw_storage_iterator<OutputIterator,T>& operator++() ~~noexcept~~;
    raw_storage_iterator<OutputIterator,T> operator++(int) ~~noexcept~~;
  };
}
```

explicit raw_storage_iterator(OutputIterator x) ~~noexcept~~;

2 *Effects*: Initializes the iterator to point to the same value to which x points.

raw_storage_iterator<OutputIterator,T>& operator*() ~~noexcept~~;

3 *Returns*: *this

raw_storage_iterator<OutputIterator,T>& operator=(const T& element);

4 *Effects*: Constructs a value from element at the location to which the iterator points.

5 *Returns*: A reference to the iterator.

raw_storage_iterator<OutputIterator,T>& operator++() ~~noexcept~~;

6 *Effects*: Pre-increment: advances the iterator and returns a reference to the updated iterator.

raw_storage_iterator<OutputIterator,T> operator++(int) ~~noexcept~~;

7 *Effects*: Post-increment: advances the iterator and returns the old value of the iterator.

20.6.11 [temporary.buffer]

template <class T> void return_temporary_buffer(T* p) ~~noexcept~~;

3 *Effects*: Deallocates the buffer to which p points.

4 *Requires*: The buffer shall have been previously allocated by get_temporary_buffer.

20.7.1.2 unique_ptr for single objects

```
namespace std {
 template <class T, class D = default_delete<T>> class unique_ptr {
 public:
  typedef see below pointer;
  typedef T element_type;
  typedef D deleter_type;

  // 20.7.1.2.1, constructors
  constexpr unique_ptr() noexcept;
  explicit unique_ptr(pointer p) noexcept;
  unique_ptr(pointer p, see below d1) noexcept;
  unique_ptr(pointer p, see below d2) noexcept;
  unique_ptr(unique_ptr&& u) noexcept;
  constexpr unique_ptr(nullptr_t) noexcept : unique_ptr() { }
  template <class U, class E>
   unique_ptr(unique_ptr<U, E>&& u) noexcept;
  template <class U>
   unique_ptr(auto_ptr<U>&& u) noexcept;

  // 20.7.1.2.2, destructor
  ~unique_ptr();

  // 20.7.1.2.3, assignment
  unique_ptr& operator=(unique_ptr&& u) noexcept;
  template <class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
  unique_ptr& operator=(nullptr_t) noexcept;
  // 20.7.1.2.4, observers
  typename add_lvalue_reference<T>::type operator*() const;
  pointer operator->() const noexcept;
  pointer get() const noexcept;
  deleter_type& get_deleter() noexcept;
  const deleter_type& get_deleter() const noexcept;
  explicit operator bool() const noexcept;

  // 20.7.1.2.5 modifiers
  pointer release() noexcept;
  void reset(pointer p = pointer()) noexcept;
  void swap(unique_ptr& u) noexcept;

  // disable copy from lvalue
  unique_ptr(const unique_ptr&) = delete;
  unique_ptr& operator=(const unique_ptr&) = delete;
 };
}
```

20.7.2.5 [util.smartptr.shared.atomic]

Concurrent access to a shared_ptr object from multiple threads does not introduce a data race if the access is done exclusively via the functions in this section and the instance is passed as their first argument.

The meaning of the arguments of type memory_order is explained in 29.3.

template<class T> bool atomic_is_lock_free(const shared_ptr<T>* p) ~~noexcept~~;
Requires: p shall not be null.
Returns: true if atomic access to *p is lock-free, false otherwise.
Throws: nothing.

template<class T>
  shared_ptr<T> atomic_load(const shared_ptr<T>* p) ~~noexcept~~;
Requires: p shall not be null.
Returns: atomic_load_explicit(p, memory_order_seq_cst).
Throws: nothing.

shared_ptr<T>
  atomic_load_explicit(const shared_ptr<T>* p, memory_order mo) ~~noexcept~~;
Requires: p shall not be null. Requires: mo shall not be memory_order_release or memory_order_acq_rel.
Returns: *p.
Throws: nothing.

template<class T>
  void atomic_store(shared_ptr<T>* p, shared_ptr<T> r) ~~noexcept~~;
Requires: p shall not be null.
Effects: atomic_store_explicit(p, r, memory_order_seq_cst).
Throws: nothing.

template<class T>
  void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo) ~~noexcept~~;
Requires: p shall not be null.
Requires: mo shall not be memory_order_acquire or memory_order_acq_rel.
Effects: p->swap(r).
Throws: nothing.

template<class T>
  shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r) ~~noexcept~~;
Requires: p shall not be null.
Returns: atomic_exchange_explicit(p, r, memory_order_seq_cst).
Throws: nothing.

template<class T> shared_ptr<T>
  atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo) ~~noexcept~~;
Requires: p shall not be null.
Effects: p->swap(r).
Returns: the previous value of *p.
Throws: nothing.

template<class T>
  bool atomic_compare_exchange_weak(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w) ~~noexcept~~;
Requires: p shall not be null.

Returns: atomic_compare_exchange_weak_explicit(p, v, w, memory_order_seq_cst, memory_-order_seq_cst).
Throws: nothing.

template<class T>
  bool atomic_compare_exchange_strong(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w) ~~noexcept~~;
Returns: atomic_compare_exchange_strong_explicit(p, v, w, memory_order_seq_cst, memory_-order_seq_cst).

template<class T>
  bool atomic_compare_exchange_weak_explicit(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w, memory_order success, memory_order failure) ~~noexcept~~;
template<class T>
  bool atomic_compare_exchange_strong_explicit(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w, memory_order success, memory_order failure) ~~noexcept~~;
Requires: p shall not be null. Requires: failure shall not be memory_order_release, memory_order_acq_rel, or stronger than success.
Effects: If *p is equivalent to *v, assigns w to *p and has synchronization semantics corresponding to the value of success, otherwise assigns *p to *v and has synchronization semantics corresponding to the value of failure.
Returns: true if *p was equivalent to *v, false otherwise.
Throws: nothing.
Remarks: two shared_ptr objects are equivalent if they store the same pointer value and share ownership.
Remarks: the weak forms may fail spuriously. See 29.6.

20.8 [function objects]

2 Header <functional> synopsis

// ....
// 20.8.13, member function adaptors:
template<class R, class T> *unspecified* mem_fn(R T::*) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...)) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) volatile) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const volatile) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) &) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const &) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) volatile &) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const volatile &) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) &&) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const &&) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) volatile &&) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const volatile &&) ~~noexcept~~;

20.8.10 [func.memfn]

template<class R, class T> *unspecified* mem_fn(R T::*) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...)) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) volatile) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const volatile) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) &) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const &) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) volatile &) ~~noexcept~~;
template<class R, class T, class... Args> *unspecified* mem_fn(R (T::*)(Args...) const volatile &) ~~noexcept~~;

```
template<class R, class T, class... Args> unspecified mem_fn(R (T::*)(Args...) &&) ~~noexcept~~;
template<class R, class T, class... Args> unspecified mem_fn(R (T::*)(Args...) const &&) ~~noexcept~~;
template<class R, class T, class... Args> unspecified mem_fn(R (T::*)(Args...) volatile &&) ~~noexcept~~;
template<class R, class T, class... Args> unspecified mem_fn(R (T::*)(Args...) const volatile &&) ~~noexcept~~;
```

1 Returns: A simple call wrapper (20.8.1) fn such that the expression fn(t, a2, ..., aN) is equivalent to INVOKE (pm, t, a2, ..., aN) (20.8.2). fn shall have a nested type result_type that is a synonym for the return type of pm when pm is a pointer to member function.

2 The simple call wrapper shall define two nested types named argument_type and result_type as synonyms for cv T* and Ret, respectively, when pm is a pointer to member function with cv-qualifier cv and taking no arguments, where Ret is pm's return type.

3 The simple call wrapper shall define three nested types named first_argument_type, second_-argument_type, and result_type as synonyms for cv T*, T1, and Ret, respectively, when pm is a pointer to member function with cv-qualifier cv and taking one argument of type T1, where Ret is pm's return type.

4 Throws: Nothing.

20.12.1 [allocator.adaptor.syn]

```
namespace std { template <class OuterAlloc, class... InnerAllocs>
class scoped_allocator_adaptor : public OuterAlloc {
public:
   void deallocate(pointer p, size_type n) ~~noexcept~~;
```

21.2.3.1 [char.traits.specializations.char]

```
namespace std {
  template<> struct char_traits<char> {
    typedef char          char_type;
    typedef int  int_type;
    typedef streamoff  off_type;
    typedef streampospos_type;
    typedef mbstate_t state_type;

    static void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;

    static int compare(const char_type* s1, const char_type* s2, size_t n) ~~noexcept~~;
    static size_t length(const char_type* s) ~~noexcept~~;
    static const char_type* find(const char_type* s, size_t n, const char_type& a) ~~noexcept~~;
    static char_type* move(char_type* s1, const char_type* s2, size_t n) ~~noexcept~~;
    static char_type* copy(char_type* s1, const char_type* s2, size_t n) ~~noexcept~~;
    static char_type* assign(char_type* s, size_t n, char_type a) ~~noexcept~~;

    static constexpr int_type not_eof(int_type c) noexcept;
    static constexpr char_type to_char_type(int_type c) noexcept;
    static constexpr int_type to_int_type(char_type c) noexcept;
    static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
    static constexpr int_type eof() noexcept;
};
}
```

21.2.3.2 [char.traits.specializations.char16_t]

```
namespace std {
  template<> struct char_traits<char> {
    typedef char          char_type;
    typedef uint_least16_t      int_type;
    typedef streamoff  off_type;
    typedef streampospos_type;
    typedef mbstate_t state_type;

    static void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;

    static int compare(const char_type* s1, const char_type* s2, size_t n) ~~noexcept~~;
    static size_t length(const char_type* s) ~~noexcept~~;
    static const char_type* find(const char_type* s, size_t n, const char_type& a) ~~noexcept~~;
    static char_type* move(char_type* s1, const char_type* s2, size_t n) ~~noexcept~~;
    static char_type* copy(char_type* s1, const char_type* s2, size_t n) ~~noexcept~~;
    static char_type* assign(char_type* s, size_t n, char_type a) ~~noexcept~~;

    static constexpr int_type not_eof(int_type c) noexcept;
    static constexpr char_type to_char_type(int_type c) noexcept;
    static constexpr int_type to_int_type(char_type c) noexcept;
    static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
    static constexpr int_type eof() noexcept;
};
}
```

21.2.3.3 [char.traits.specializations.char32_t]

```
namespace std {
  template<> struct char_traits<char> {
    typedef char          char_type;
    typedef uint_least32_t      int_type;
    typedef streamoff  off_type;
    typedef streampospos_type;
    typedef mbstate_t state_type;

    static void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;

    static int compare(const char_type* s1, const char_type* s2, size_t n) ~~noexcept~~;
    static size_t length(const char_type* s) ~~noexcept~~;
    static const char_type* find(const char_type* s, size_t n, const char_type& a) ~~noexcept~~;
    static char_type* move(char_type* s1, const char_type* s2, size_t n) ~~noexcept~~;
    static char_type* copy(char_type* s1, const char_type* s2, size_t n) ~~noexcept~~;
    static char_type* assign(char_type* s, size_t n, char_type a) ~~noexcept~~;

    static constexpr int_type not_eof(int_type c) noexcept;
    static constexpr char_type to_char_type(int_type c) noexcept;
    static constexpr int_type to_int_type(char_type c) noexcept;
    static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
    static constexpr int_type eof() noexcept;
};
}
```

21.2.3.4 [char.traits.specializations.wchar.t]

```
namespace std {
  template<> struct char_traits<char> {
    typedef wchar_t    char_type;
    typedef wint_t       int_type;
    typedef streamoff  off_type;
    typedef streampospos_type;
    typedef mbstate_t state_type;


    static void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;

    static int compare(const char_type* s1, const char_type* s2, size_t n) ~~noexcept~~;
    static size_t length(const char_type* s) ~~noexcept~~;
    static const char_type* find(const char_type* s, size_t n, const char_type& a) ~~noexcept~~;
    static char_type* move(char_type* s1, const char_type* s2, size_t n) ~~noexcept~~;
    static char_type* copy(char_type* s1, const char_type* s2, size_t n) ~~noexcept~~;
    static char_type* assign(char_type* s, size_t n, char_type a) ~~noexcept~~;

    static constexpr int_type not_eof(int_type c) noexcept;
    static constexpr char_type to_char_type(int_type c) noexcept;
    static constexpr int_type to_int_type(char_type c) noexcept;
    static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
    static constexpr int_type eof() noexcept;
};
}
```

21.3 [string.classes]

Header <string> synopsis

```
#include <initializer_list>
namespace std {
// 21.4, basic_string:
// ....
template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT,traits,Allocator>& lhs, const basic_string<charT,traits,Allocator>& rhs)
~~noexcept~~;

template<class charT, class traits, class Allocator>
  bool operator==(const charT* lhs, const basic_string<charT,traits,Allocator>& rhs) ~~noexcept~~;
template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT,traits,Allocator>& lhs, const charT* rhs) ~~noexcept~~;

template<class charT, class traits, class Allocator> bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
const basic_string<charT,traits,Allocator>& rhs) ~~noexcept~~;

template<class charT, class traits, class Allocator>
  bool operator!=(const charT* lhs, const basic_string<charT,traits,Allocator>& rhs) ~~noexcept~~;
template<class charT, class traits, class Allocator>
  bool operator!=(const basic_string<charT,traits,Allocator>& lhs, const charT* rhs) ~~noexcept~~;

template<class charT, class traits, class Allocator>
  bool operator< (const basic_string<charT,traits,Allocator>& lhs, const basic_string<charT,traits,Allocator>& rhs)
~~noexcept~~;

template<class charT, class traits, class Allocator>
  bool operator< (const basic_string<charT,traits,Allocator>& lhs, const charT* rhs) ~~noexcept~~;
template<class charT, class traits, class Allocator>
  bool operator< (const charT* lhs, const basic_string<charT,traits,Allocator>& rhs) ~~noexcept~~;

template<class charT, class traits, class Allocator>
```

```
  bool operator> (const basic_string<charT,traits,Allocator>& lhs, const basic_string<charT,traits,Allocator>& rhs)
~~noexcept~~;

template<class charT, class traits, class Allocator>
  bool operator> (const basic_string<charT,traits,Allocator>& lhs, const charT* rhs) ~~noexcept~~;
template<class charT, class traits, class Allocator>
  bool operator> (const charT* lhs, const basic_string<charT,traits,Allocator>& rhs) ~~noexcept~~;

template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT,traits,Allocator>& lhs, const basic_string<charT,traits,Allocator>& rhs)
~~noexcept~~;

template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT,traits,Allocator>& lhs, const charT* rhs) ~~noexcept~~;
template<class charT, class traits, class Allocator>
  bool operator<=(const charT* lhs, const basic_string<charT,traits,Allocator>& rhs) ~~noexcept~~;

template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT,traits,Allocator>& lhs, const basic_string<charT,traits,Allocator>& rhs)
~~noexcept~~;

template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT,traits,Allocator>& lhs, const charT* rhs) ~~noexcept~~;
template<class charT, class traits, class Allocator>
  bool operator>=(const charT* lhs,const basic_string<charT,traits,Allocator>& rhs) ~~noexcept~~;

// 21.4.8.8: swap
template<class charT, class traits, class Allocator>
void swap(basic_string<charT,traits,Allocator>& lhs, basic_string<charT,traits,Allocator>& rhs) ~~noexcept~~;
```

21.4 [basic.string]

```
namespace std {
template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT> >
class basic_string {
public:
    //...
    // 21.4.5 element access:
    const_reference   operator[](size_type pos) const ~~noexcept~~;
    reference         operator[](size_type pos) ~~noexcept~~;
    const_reference   at(size_type n) const;
    reference         at(size_type n);
    const charT& front() const ~~noexcept~~;
    charT& front() ~~noexcept~~;
    const charT& back() const ~~noexcept~~;
    charT& back() ~~noexcept~~;

    // 21.4.6 modifiers:
    // ...
    void pop_back() ~~noexcept~~;

   //...
   void swap(basic_string& str) ~~noexcept~~;

   // 21.4.7 string operations:
   const charT* c_str() const noexcept;
   const charT* data() const noexcept;
   allocator_type get_allocator() const noexcept;
```

```
  size_type find (const basic_string& str, size_type pos = 0) const noexcept;
  size_type find (const charT* s, size_type pos, size_type n) const ~~noexcept~~;
  size_type find (const charT* s, size_type pos = 0) const ~~noexcept~~;
  size_type find (charT c, size_type pos = 0) const noexcept;

  size_type rfind(const basic_string& str, size_type pos = npos) const noexcept;
  size_type rfind(const charT* s, size_type pos, size_type n) const ~~noexcept~~;
  size_type rfind(const charT* s, size_type pos = npos) const ~~noexcept~~;
  size_type rfind(charT c, size_type pos = npos) const noexcept;

  size_type find_first_of(const basic_string& str, size_type pos = 0) const noexcept;
  size_type find_first_of(const charT* s, size_type pos, size_type n) const ~~noexcept~~;
  size_type find_first_of(const charT* s, size_type pos = 0) const ~~noexcept~~;
  size_type find_first_of(charT c, size_type pos = 0) const noexcept;

  size_type find_last_of (const basic_string& str, size_type pos = npos) const noexcept;
  size_type find_last_of (const charT* s, size_type pos, size_type n) const ~~noexcept~~;
  size_type find_last_of (const charT* s, size_type pos = npos) const ~~noexcept~~;
  size_type find_last_of (charT c, size_type pos = npos) const noexcept;

  size_type find_first_not_of(const basic_string& str, size_type pos = 0) const noexcept;
  size_type find_first_not_of(const charT* s, size_type pos, size_type n) const ~~noexcept~~;
  size_type find_first_not_of(const charT* s, size_type pos = 0) const ~~noexcept~~;
  size_type find_first_not_of(charT c, size_type pos = 0) const noexcept;

  size_type find_last_not_of (const basic_string& str, size_type pos = npos) const noexcept;
  size_type find_last_not_of (const charT* s, size_type pos, size_type n) const ~~noexcept~~;
  size_type find_last_not_of (const charT* s, size_type pos = npos) const ~~noexcept~~;
  size_type find_last_not_of (charT c, size_type pos = npos) const noexcept;

  basic_string substr(size_type pos = 0, size_type n = npos) const;
  int compare(const basic_string& str) const noexcept;
  int compare(size_type pos1, size_type n1, const basic_string& str) const;
  int compare(size_type pos1, size_type n1, const basic_string& str, size_type pos2, size_type n2) const;
  int compare(const charT* s) const ~~noexcept~~;
  int compare(size_type pos1, size_type n1, const charT* s) const;
  int compare(size_type pos1, size_type n1, const charT* s, size_type n2) const;
 };
}
```

21.4.5 [string.access]

const_reference operator[](size_type pos) const ~~noexcept~~;
reference operator[](size_type pos) ~~noexcept~~;

Requires: pos <= size().

Returns: *(begin() + pos) if pos < size(), otherwise a reference to an object of type T with value charT(); the referenced value shall not be modified.

Complexity: constant time.

Throws: Nothing.

21.4.6.8 [string::swap]

void swap(basic_string<charT,traits,Allocator>& s) ~~noexcept~~;

Throws: Nothing.

Postcondition: *this contains the same sequence of characters that was in s, s contains the same sequence of characters that was in *this.

Complexity: constant time.

23.3.3 [forwardlist]

```
namespace std {
  template <class T, class Allocator = allocator<T> >
    class forward_list {
    public:
      // ...
      // 23.3.3.4 modifiers:
      // ...
      iterator erase_after(const_iterator position) ~~noexcept~~;
      iterator erase_after(const_iterator position, iterator last) ~~noexcept~~;

      // 23.3.3.5 forward_list operations:
      void splice_after(const_iterator position, forward_list<T,Allocator>&& x) ~~noexcept~~;
      void splice_after(const_iterator position, forward_list<T,Allocator>&& x, const_iterator i) ~~noexcept~~;
  };
}
```

23.3.4.5 [forwardlist.modifiers]

iterator erase_after(const_iterator position) ~~noexcept~~;

19 Requires: The iterator following position is dereferenceable.

20 Effects: Erases the element pointed to by the iterator following position.

21 Returns: An iterator pointing to the element following the one that was erased, or end() if no such element exists.

? Throws: Nothing.

iterator erase_after(const_iterator position, iterator last) ~~noexcept~~;

22 Requires: All iterators in the range (position,last) are dereferenceable.

23 Effects: Erases the elements in the range (position,last).

23 Returns: last.

? Throws: Nothing.

23.3.4.6 [forwardlist.ops]

void splice_after(const_iterator position, forward_list<T,Allocator>&& x) ~~noexcept~~;

Requires: position is before_begin() or is a dereferenceable iterator in the range [begin(),end()). &x != this.

Effects: Inserts the contents of x after position, and x becomes empty. Pointers and references to the moved elements of x now refer to those same elements but as members of *this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this, not into x.

Throws: Nothing.

Complexity: O(distance(x.begin(),x.end()))

void splice_after(const_iterator position, forward_list<T,Allocator>&& x, const_iterator i) ~~noexcept~~;

Requires: position is before_begin() or is a dereferenceable iterator in the range [begin(),end()). The iterator following i is a dereferenceable iterator in x.

Effects: Inserts the element following i into *this, following position, and removes it from x. The result is unchanged if position == i or position == ++i. Pointers and references to *i continue to refer to the same element but as a member of *this. Iterators to *i (including i itself) continue to refer to the same element, but now behave as iterators into *this, not into x.

Throws: Nothing.

Complexity: O(1)

23.3.4 [list]

```
namespace std {
 template <class T, class Allocator = allocator<T> >
  class list {
  public:
   // ...
   // 23.3.4.3 modifiers:
   template <class... Args> void emplace_front(Args&&... args);
   void pop_front() ~~noexcept~~;
   template <class... Args> void emplace_back(Args&&... args);
   void push_front(const T& x);
   void push_front(T&& x);
   void push_back(const T& x);
   void push_back(T&& x);
   void pop_back() ~~noexcept~~;

   // ...

   iterator erase(const_iterator position) ~~noexcept~~;
   iterator erase(const_iterator position, const_iterator last) ~~noexcept~~;
   void swap(list<T,Allocator>&);
   void clear() noexcept;

   // 23.3.4.4 list operations:
   void splice(const_iterator position, list<T,Allocator>& x) ~~noexcept~~;
   void splice(const_iterator position, list<T,Allocator>&& x) ~~noexcept~~;
   void splice(const_iterator  position, list<T,Allocator>& x, const_iterator i) ~~noexcept~~;
   void splice(const_iterator position, list<T,Allocator>&& x, const_iterator i) ~~noexcept~~;
   void splice(const_iterator position, list<T,Allocator>& x, const_iterator first, const_iterator last) ~~noexcept~~;
   void splice(const_iterator position, list<T,Allocator>&& x, const_iteratorfirst, const_iterator last) ~~noexcept~~;
  };
}
```

23.3.5.4 [list.modifiers]

iterator erase(const_iterator position) ~~noexcept~~;
iterator erase(const_iterator first, const_iterator last) ~~noexcept~~;

void pop_front() ~~noexcept~~;
void pop_back() ~~noexcept~~;
void clear() noexcept;

Effects: Invalidates only the iterators and references to the erased elements.

Throws: Nothing.

Complexity: Erasing a single element is a constant time operation with a single call to the destructor of T. Erasing a range in a list is linear time in the size of the range and the number of calls to the destructor of type T is exactly equal to the size of the range.

23.3.5.5 [list.ops]

void splice(const_iterator position, list<T,Allocator>& x) ~~noexcept~~;
void splice(const_iterator position, list<T,Allocator>&& x) ~~noexcept~~;

3 Requires: &x != this.

4 Effects: Inserts the contents of x before position and x becomes empty. Pointers and references to the moved elements of x now refer to those same elements but as members of *this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this, not into x.

? Throws: Nothing.

5 Complexity: Constant time.

void splice(const_iterator position, list<T,Allocator>& x, const_iterator i) ~~noexcept~~;
void splice(const_iterator position, list<T,Allocator>&& x, const_iterator i) ~~noexcept~~;

6 Effects: Inserts an element pointed to by i from list x before position and removes the element from x. The result is unchanged if position == i or position == ++i. Pointers and references to *i continue to refer to this same element but as a member of *this. Iterators to *i (including i itself) continue to refer to the same element, but now behave as iterators into *this, not into x.

7 Requires: i is a valid dereferenceable iterator of x.

? Throws: Nothing.

8 Complexity: Constant time.

void splice(const_iterator position, list<T,Allocator>& x, const_iterator first, const_iterator last) ~~noexcept~~;
void splice(const_iterator position, list<T,Allocator>&& x, const_iterator first, const_iterator last) ~~noexcept~~;

9 Effects: Inserts elements in the range [first,last) before position and removes the elements from x.

10 Requires: [first, last) is a valid range in x. The result is undefined if position is an iterator in the range [first,last). Pointers and references to the moved elements of x now refer to those same elements but as members of *this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this, not into x.

? Throws: Nothing.

11 Complexity: Constant time if &x == this; otherwise, linear time.

27.5.4 [ios]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
    class basic_ios : public ios_base {
      // ....
    protected:
      basic_ios();
      void init(basic_streambuf<charT,traits>* sb);
      void move(basic_ios& rhs);
      void move(basic_ios&& rhs);
      void swap(basic_ios& rhs) noexcept;
      void set_rdbuf(basic_streambuf<charT, traits>* sb) ~~noexcept~~;
  };
}
```

27.5.4.2 [basic.ios.members]

void set_rdbuf(basic_streambuf<charT, traits>* sb) ~~noexcept~~;

Requires: sb != nullptr.

Effects: Associates the basic_streambuf object pointed to by sb with this stream without calling clear().

Postconditions: rdbuf() == sb.

Throws: Nothing.